19-函数式编程之不变性：怎样保证我的代码不会被别人破坏？

你好！我是郑晔。

经过前两讲的介绍，你已经认识到了函数式编程的能力，函数以及函数之间的组合很好地体现出了函数式编程的巧妙之处。不过，我们在讲编程范式时说过，学习编程范式不仅要看它提供了什么，还要看它约束了什么。这一讲，我们就来看看函数式编程对我们施加的约束。

在软件开发中，有一类Bug是很让人头疼的，就是你的代码怎么看都没问题，可是运行起来就是出问题了。我曾经就遇到过这样的麻烦，有一次我用C写了一个程序，怎么运行都不对。我翻来覆去地看自己的代码，看了很多遍都没发现问题，不得已，只能一步一步跟踪代码。最后，我发现我的代码调用到一个程序库时，出现了与预期不符的结果。

这个程序库是其他人封装的，我只是拿过来用。按理说，我调用的这个函数逻辑也不是特别复杂，不应该出现什么问题。不过，为了更快定位问题，我还是打开了这个程序库的源代码。经过一番挖掘，我发现在这个函数底层实现中，出现了一个全局变量。

分析之后，我发现正是这个全局变量引起了这场麻烦，因为在我的代码执行过程中，有别的程序会调用另外的函数，修改这个全局变量的值，最终，导致了我的程序执行失败。从表面上看，我调用的这个函数和另外那个函数八竿子都打不到，但是，它们却通过一个底层的全局变量，产生了相互的影响。

这就是一类非常让人头疼的Bug。有人认为这是全局变量使用不当造成的，在Java设计中，甚至取消了全局变量，但类似的问题并没有因此减少，只是以不同面貌展现出来而已，比如，static 变量。

那么造成这类问题的真正原因是什么呢？**真正原因就在于变量是可变的**。

变之殇

你可能会好奇，难道变量不就应该是变的吗？为了更好地理解这一类问题，我们来看一段代码：

```
class Sample1 {
  private static final DateFormat format =
      new SimpleDateFormat("yyyy.MM.dd");

  public String getCurrentDateText() {
    return format.format(new Date());
  }
}
```

如果你不熟悉JDK的SimpleDateFormat，你可能会觉得这段代码看上去还不错。然而，这段代码在多线程环境下就会出问题。正确的用法应该是这样：

```
public class Sample2 {
  public String getCurrentDateText() {
    DateFormat format = new SimpleDateFormat("yyyy.MM.dd");
    return format.format(new Date());
  }
```

```
}
```

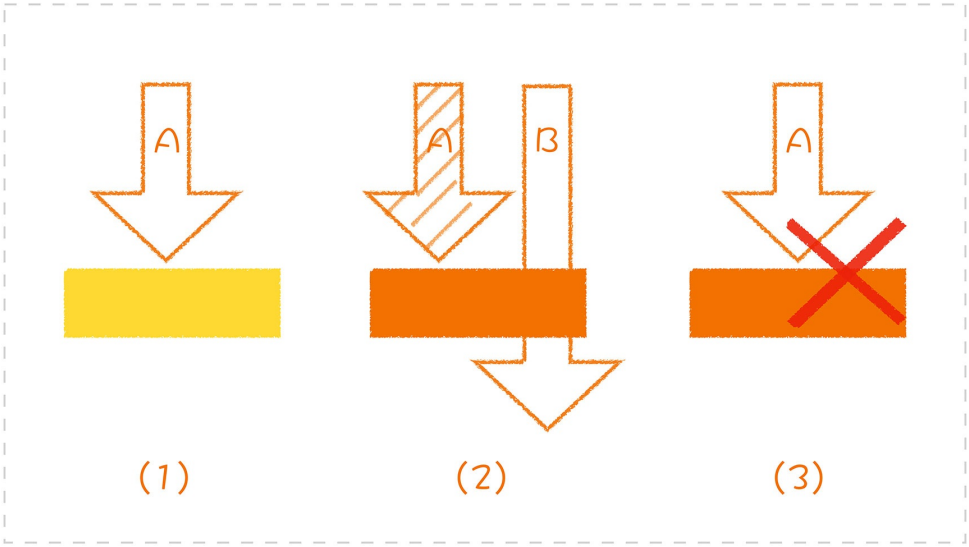
两段代码最大的区别就在于，SimpleDateFormat在哪里构建。一个是被当作了一个字段，另一个则是在函数内部构建出来。这两种不同做法的根本差别就在于，SimpleDateFormat对象是否共享。

为什么这个对象共享会有问题呢？翻看format方法的源码，你会发现这样一句：

```
calendar.setTime(date);
```

这里的calendar是SimpleDateFormat这个类的一个字段，正是因为在format的过程中修改了calendar字段，所以，它才会出问题。

我们来看看这种问题是怎么出现的，就像下面这张图看到的：



- A线程把变量的值修改成自己需要的值；
- 这时发生线程切换，B线程开始执行，将变量的值修改成它所需要的值；
- 线程切换回来，A线程继续执行，但此时变量已经不是自己设置的值了，所以，执行会出错。

回到SimpleDateFormat上，问题是一样的，calendar就是那个共享的变量。一个线程刚刚设置的值，可能会被另外一个线程修改掉，因此会造成结果的不正确。而在Sample2的写法中，通过每次创建一个新的SimpleDateFormat对象，我们将二者之间的共享解开，规避了这个问题。

那如果我还是想按照Sample1的写法写，SimpleDateFormat这个库应该怎么改写呢？可能你会想，SimpleDateFormat的作者没写好，如果换我写，我就会给它加上一个同步（synchronized）或者加上锁（Lock）。你甚至都没有注意，你轻易地将多线程的复杂性引入了进来。还记得我在分离关注点那节讨论

的问题吗，多线程是另外一个关注点，能少用，尽量少用。

一个更好的办法是将calendar变成局部变量，这样一来，不同线程之间共享变量的问题就得到了根本的解决。但是，这类非常头疼的问题在函数式编程中却几乎不存在，这就依赖于函数式编程的不变性。

不变性

函数式编程的不变性主要体现在值和纯函数上。值，你可以将它理解为一个初始化之后就不再改变的量，换句话说，当你使用一个值的时候，值是不会变的。纯函数，是符合下面两点的函数：

- 对于相同的输入，给出相同的输出；
- 没有副作用。

把值和纯函数合起来看，**值保证不会显式改变一个量，而纯函数保证的是，不会隐式改变一个量**。

我们说过，函数式编程中的函数源自数学中的函数。在这个语境里，函数就是纯函数，一个函数计算之后是不会产生额外的改变的，而函数中用到的一个一个量就是值，它们是不会随着计算改变的。所以，在函数式编程中，计算天然就是不变的。

正是由于不变性的存在，我们在前面遇到的那些问题也就不再是问题了。一方面，如果你拿到一个量，这次的值是1，下一次它还是1，我们完全不用担心它会改变。另一方面，我们调用一个函数，传进去同样的参数，它保证给出同样的结果，行为是完全可以预期的，不会碰触到其他部分。即便是在多线程的情况下，我们也不必考虑同步的问题，后续一系列的问题也就不存在了。

这与我们习惯的方式有着非常大的区别，因为传统方式的基础是面向内存单元的，改来改去甚至已经成为了程序员的本能。所以，我们对counter = counter + 1这种代码习以为常，而初学编程的人总会觉得这在数学上是不成立的。

在之前的讨论中，我们说过，传统的编程方式占优的地方是执行效率，而现如今，这个优点则越来越不明显，反而是因为到处可变而带来了更多的问题。相较之下，我们更应该在现在的设计中，考虑借鉴函数式编程的思路，把不变性更多地应用在我们的代码之中。

那怎么应用呢？首先是值。我们可以编写不变类，就是对象一旦构造出来就不能改变，Java程序员最熟悉的不变类应该就是String类，怎样编写不变类呢？

- 所有的字段只在构造函数中初始化；
- 所有的方法都是纯函数；
- 如果需要有改变，返回一个新的对象，而不是修改已有字段。

前面两点可能还好理解，最后一点，我们可以看一下Java String类的replace方法签名：

```
String replace(char oldChar, char newChar);
```

在这里，我们会用一个新的字符（newChar）替换掉这个字符串中原有的字符（oldChar），但我们并不是直接修改已有的这个字符串，而是创建一个新的字符串对象返回。这样一来，使用原来这个字符串的类并不用担心自己引用的内容会随之变化。

有了这个基础，等我们后面学习领域驱动设计的时候，你就很容易理解值对象（Value Object）是怎么回事了。

我们再来看纯函数。**编写纯函数的重点是，不修改任何字段，也不调用修改字段内容的方法**。因为在实际的工作中，我们使用的大多数都是传统的程序设计语言，而不是严格的函数式编程语言，不是所有用到的量都是值。所以，站在实用性的角度，如果要使用变量，就使用局部变量。

还有一个实用性的编程建议，就是使用语法中不变的修饰符，比如，Java就尽可能多使用final，C/C++就多写const。无论是修饰变量还是方法，它们的主要作用就是让编译器提醒你，要多从不变的角度思考问题。

当你有了用不变性思考问题的角度，你会发现之前的很多编程习惯是极其糟糕的，比如，Java程序员最喜欢写的setter，它就是提供了一个接口，修改一个对象内部的值。

不过，纯粹的函数式编程是很困难的，我们只能把编程原则设定为**尽可能编写不变类和纯函数**。但仅仅是这么来看，你也会发现，自己从前写的很多代码，尤其是大量负责业务逻辑处理的代码，完全可以写成不变的。

绝大多数涉及到可变或者副作用的代码，应该都是与外部系统打交道的。能够把大多数代码写成不变的，这已经是一个巨大的进步，也会减少许多后期维护的成本。

而正是不变性的优势，有些新的程序设计语言默认选项不再是变量，而是值。比如，在Rust里，你这么声明的是一个值，因为一旦初始化了，你将无法修改它：

```
let result = 1;
```

而如果你想声明一个变量，必须显式地告诉编译器：

```
let mut result = 1;
```

Java也在尝试将值类型引入语言，有一个专门的Valhalla 项目就是做这个的。你也看到了，不变性，是减少程序问题的一个重要努力方向。

现在回过头来看编程范式那一讲里说的约束：

> 函数式编程，限制使用赋值语句，它是对程序中的赋值施加了约束。

理解了不变性，你应该知道这句话的含义了，一旦初始化好一个量，就不要随便给它赋值了。

总结时刻

今天，我们讲了无论是全局变量、还是多线程，变化给程序设计带来了很多麻烦，然后我们还分析了这类问题的成因。

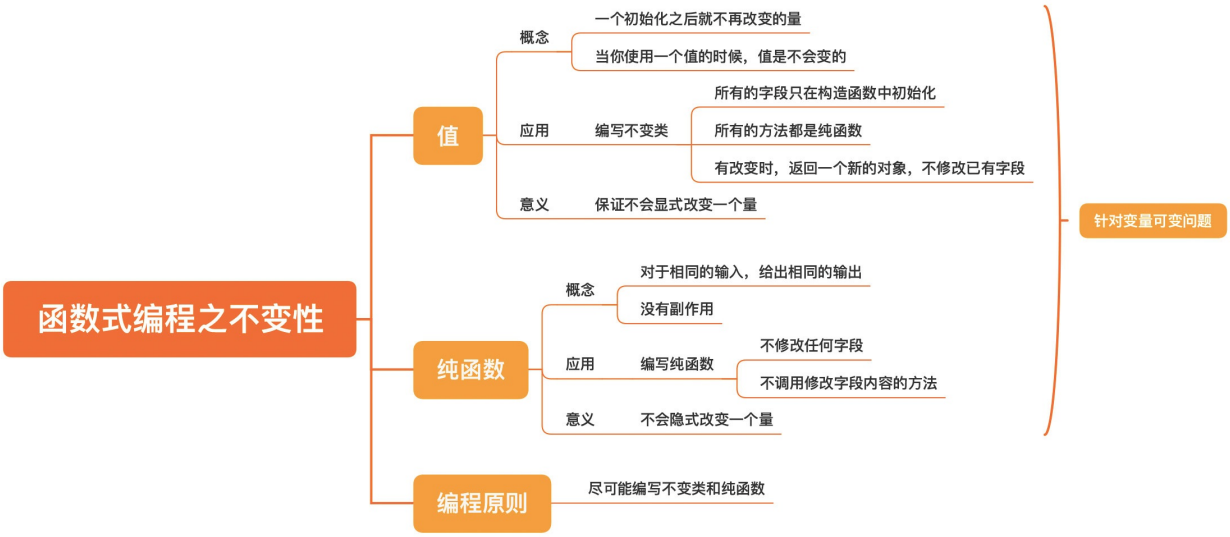
然而，这类问题在函数式编程中并不存在。其中，重要的原因就是函数式编程的不变性。函数式编程的不变性主要体现在它的值和纯函数上。深入学习函数式编程时，你会遇到的与之相关的各种说法：无副作用、无状态、引用透明等等，其实都是在讨论不变性。

即便使用传统的程序设计语言，我们也可以从中借鉴一些编程的方法。比如，编写不变类、编写纯函数、尽量使用不变的修饰符等等。

经过了这三讲的介绍，相信你已经对函数式编程有了很多认识，不过，我只是把设计中最常用的部分给你做了一个介绍，这远远不是函数式编程的全部。就算Java这种后期增补的函数式编程的语言，其中也包含了惰性求值、Optional等诸多内容，值得你去深入了解。不过我相信有了前面知识的铺垫，你再去学习函数式编程其他相关内容，难度系数就会降低一些。

关于编程范式的介绍，我们就告一段落，下一讲，我们开始介绍设计原则。

如果今天的内容你只能记住一件事，那请记住：**尽量编写不变类和纯函数。**



极客时间

思考题

最后，我想请你去了解一下Event Sourcing，结合今天的内容，谈谈你对它的理解。欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- 阳仔 2020-07-08 09:45:30
  变化是软件开发的永恒主题，所以在编码实践上尽量的编写不变的纯函数和类，将变化的粒度控制到最小

作者回复2020-07-08 11:50:32
变化是需求层面的不得已，不变是代码层面的努力控制。

- 赵冲 2020-07-08 08:31:27
尤其是大量负责业务逻辑处理的代码，完全可以写成不变的。这句话不太理解，老师可以举个例子吗？
事件溯源，对比一般的CRUD，就是没有修改，只有不断的插入值不同的同一条记录，下次修改时，在最新一条基础上修改值后再插入一条最新的。有点类似Java String 的处理方式，修改是生成另一个对象。

- NIU 2020-07-08 08:03:07
初始化后不会改变的“值”就是常量吗？

作者回复2020-07-08 10:26:52
常量一般是预先确定的，而值是在运行过程中生成的。